УДК 004.415.53+004.832.23

А.В. Колчин, С.В. Потиенко
Институт кибернетики им. В.М. Глушкова НАН Украины, Украина
пр. академика Глушкова, 40, г. Киев, 03680

МЕТОД ГЕНЕРАЦИИ ТЕСТОВЫХ ДАННЫХ ПО ИСХОДНОМУ КОДУ JAVA ПРОГРАММ

A. Kolchin, S. Potiyenko
V.M. Glushkov Institute of Cybernetics of the National Academy of Sciences of Ukraine, Ukraine
Academik Glushkov av., 40, Kyiv, 03680

A METHOD OF TEST DATA GENERATION FROM SOURCE CODE OF JAVA PROGRAMS

Цель предлагаемого метода – повышение эффективности автоматической генерации и минимизации множества тестовых данных для обеспечения покрытия исходного кода Java программ. Рассматриваются различные виды покрытия, способы абстрактной интерпретации и редукции пространства поиска. В основу метода положены формальные методы анализа поведения модели.
Ключевые слова: тестирование, редукция пространства поиска.

The objective of proposed method is to increase efficiency of automatic generation and minimization of test data set needed to guarantee coverage of source code of Java programs. Different kinds of coverage, methods of abstract interpretation and state-space reduction are discussed. The basis of the proposed method is formal methods of model behavior analysis.
Key words: testing, state-space search reduction.

Введение

Данная работа посвящена автоматической генерации тестов из промышленного программного кода. Обычно такие тесты позволяют проверить только неявные требования, например, отсутствие зависаний или неожиданных исключительных ситуаций во время выполнения теста, но не обнаружение несоответствия реализации системы и ее спецификации. В современной программной индустрии актуальна проблема модернизации устаревшего кода (legacy code), по сути, не имеющего спецификаций. Часто исходный код является единственным достоверным источником знаний о функционировании системы ввиду многих причин – ее разработчики не доступны, проектная документация не полна или отсутствует, а сопровождающий персонал имеет только поверхностные знания. Тем не менее, стремительное развитие ИТ требует модификаций и модернизаций, например, в последнее время приобретает популярность описание кода на базе бизнес-правил. Основное преимущество бизнес-правил в упрощении как понимания работы системы, так и дальнейших модификаций. При такой модернизации сначала выполняется переписывание кода (на современный язык программирования, с использованием новых технологий) при сохранении исходной функциональности. Таким образом, автоматически сгенерированный набор тестов может быть использован для проверки сохранения функциональности на новой реализации.

Непосредственная задача предлагаемого метода – эффективно вычислить набор входных тестовых данных и последовательностей входных сигналов, а так же минимизировать такой набор для обеспечения заданного критерия покрытия.

Обзор существующих методов

Построение тестов из кода для языка Java – популярная тема исследований и разработок [1–4]. Автоматическая генерация эффективных тестовых наборов открывает перспективу исчерпывающего тестирования программного обеспечения в разумные

сроки и умеренный бюджет. К наиболее развитым современным подходам к генерации тестов по исходному коду можно отнести стохастические методы (random testing), систематические методы символьного выполнения (symbolic execution) и методы поиска оптимального приближения (search-based) [1, 2]. В качестве примера успешного инструмента применительно к языку Java среди symbolic execuction методов можно привести разработку Symbolic PathFinder [3] от NASA, из search-based – инструмент EvoSuite[4], который использует генетические алгоритмы для эволюции тестов. У каждого инструмента есть свои отличительные преимущества (см., например, обзоры [1, 2]), однако все они имеют проблемы с размером задач. Рассмотрим пример простой программы на рис.1(а). Symbolic PathFinder показал экспоненциальный рост множества состояний с увеличением количества параметров, EvoSuite за 10 минут работы не удалось покрыть (см. рис. 1.б) присваивание ‘res = 1’ для случая с пятью параметрами.

```
public int func(int p1, int p2, …){
   int res = 0, r1 = 0, r2 = 0, …;
   if(p1 == 1)
      r1 = 1;
   if(p2 == 1 && r1 == 1)
      r2 = 1;
   ...
   if(pn == 1 && rn-1 == 1)
      res = 1;
    return res;
}
```

Рис. 1(а). Пример Java программы

```
Тест №1:
func((-684),1147,1147,(-1),1);

Тест №2:
func(1, 1, 1, (-1), (-1));

Тест№3:
func((-1),(-1),(-1469),1,(-83));
```

Рис. 1(б). EvoSuite тесты

В большинстве случаев инструменты используют критерии покрытия операторов (statement coverage) и ветвей потока управления (branch coverage). Однако такие критерии малоэффективны при обнаружении ошибок; с другой стороны, покрытие всех путей неосуществимо на практике, ведь в общем случае программы могут иметь бесконечно много путей [5].

Постановка задачи

Перед авторами была поставлена достаточно амбициозная задача – в идеальном случае, это автоматическая генерация тестов, обеспечивающих 100% покрытие legacy-кода на языке Java. При этом предполагается, что это будет промышленный код, содержащий сотни тысяч строк. Перечислим основные подзадачи на пути достижения поставленной цели данной работы:

- трансляция конструкций языка Java в формальную модель требует построения разного уровня адекватных абстракций и методов их уточнений;
- анализ достижимости, к которому сводится задача обеспечения покрытия. Ввиду неразрешимости в общем случае, мы будем прибегать к огрублениям и упрощениям, например, использовать технику абстракции предикатов для типов данных с широким (бесконечным) диапазоном допустимых значений;
- сокращение комбинаторного взрыва, как составляющая проблемы достижимости, заслуживает отдельного рассмотрения. Ведь даже при упрощении исходного кода к модели с булевыми атрибутами, количество ее состояний растет экспоненциально с увеличением количества атрибутов, а так как поставленная задача требует обрабатывать сотни тысяч строк кода, решение этой проблемы носит определяющий характер;
- вычисление исходных ограничений для теста (и, аналогично ограничений на значения параметров промежуточных сигналов), хоть и имеет теоретическое обоснование [6–8] и практическое решение [9], требует значительного повышения эффективности, ведь

предполагаемая последовательность выполненного кода в среднем будет превосходить тысячу строк для каждого теста;
- минимизация количества результирующих тестов и повышение их качества.

Выполнение теста будет занимать некоторое время, и из практических соображений множество тестов целесообразно содержать настолько минимизированным, насколько позволяют ограничения выполнения проекта. При этом повышаются требования к выбранным тестам: каждому из них желательно иметь смысловое содержание, например, покрывать поведение некоторой функциональности от ее начала до некоторого логического завершения. Эта особенность обычно не выдерживается при существующих подходах – автоматически сгенерированные тесты, как правило, получаются несвязными и обрывающимися в произвольной точке (например, некоторое новое значение вычислено, но к его отображению трасса не привела ввиду завершения по обработке несвязанной исключительной ситуации).

Идеального решения поставленной задачи не существует ввиду многих объективных теоретических причин, но с другой стороны индустрия ИТ требует эффективного практического решения.

Описание метода

Предложенный метод относится к классу систематических symbolic execution методов (точнее, symbolic modeling) и имеет составляющие: трансляцию, суть которой сводится к построению адекватной формальной модели, и model checking [10] как эффективный инструмент поиска трасс.

Транслятор строит формальную модель и данные для обратной трансляции. Включает синтаксический разбор исходного кода (подмножество Java SE 8 [11]), статический анализ и преобразования потока управления и данных. В результате синтаксического разбора исходного кода формируется информация о так называемых def-use ветвях потока данных [5], а также строится граф потока управления (с учетом наследования и других принципов ООП). Его вершины мы называем событиями, которые представляют операторы исходного кода и делятся на следующие типы:

- assignment – присваивание;
- decision – ветвление по условию;
- loop – цикл с условием завершения;
- call – вызов функции (метода);
- return – возврат из функции;
- goto – безусловный переход (практически не используется);
- exception – исключение (рассмотрение выходит за рамки данной статьи);
- output – вывод данных;
- input – ввод данных.

Выражения представляются в виде абстрактного синтаксического дерева (AST) и в дальнейшем подлежат преобразованию в язык формальной модели. Вызовы методов выносятся из выражений в отдельные события в порядке их выполнения. В самих же выражениях они заменяются на серийные номера созданных событий. Например,

```
String name = staff.getPerson(i).getName();
```

отобразится как последовательность событий:

```
01: call staff.getPerson(i);
02: call #01.getName();
03: assignment name := #02;
```

Тело метода хранится как отдельный граф (body), на который ссылаются события типа call. При переводе выражений вызовов из AST в язык формальной модели мы генерируем отдельные присваивания для каждого параметра (рекурсия не поддерживается).

Не редко встречаются обращения к методам классов, отсутствующих в анализируемом коде. Это относится как к внешним, так и к стандартным библиотекам Java. Некоторые наиболее популярные методы классов из стандартных библиотек представлены на языке формальной модели (например, коллекция ArrayList), однако многие вызовы остаются неинтерпретированными. Для них допускается абстракция – call-событие заменяется на input-событие, представляющее возвращаемое значение вызванного метода как любое значение из вне. Это возможно благодаря внедренным методам символьного моделирования, позволяющим работать с атрибутами, не имеющими конкретных значений.

Трансляция графа потока управления и AST в формальную модель происходит в 3 этапа: трансформация объектов, фильтрация кода и определение типов.

Трансформация объектов. Все нестатические поля классов представляются в формальной модели в виде массивов (поля-массивы – в виде многомерных массивов), а ссылки на экземпляры классов – в виде индексов этих массивов. Т.е. обращение к полю конкретного объекта транслируется в оператор доступа к элементу массива по индексу. Вызов метода для объекта в модели является вызовом функции, где индекс объекта передается первым параметром, а остальные параметры сдвигаются на одну позицию. Пример построения формальной модели представлен на рис.2:

<table>
<tr><th>Java код</th><th>Формальная модель</th></tr>
<tr><td><pre>public class Person {
  private String name;
  public String setName(String name){
    this.name = name;
  }
}
...
Person p = new Person();
p.setName("Bear Grylls");</pre></td><td><pre>environment: {
  type T1: {sBear_Grylls, OTHER_T1};
  Person_counter: int;
  p: int;
  Person_name: array of T1;
}
...
function: setName;
scope: Person;
parameters: {
  PS_this: int; PS_name: T1;
}
body: {
01: assignment
    Person_name[PS_this] := PS_name;
}
...
10: assignment
    Person_counter:= Person_counter+1;
11: assignment
    p := Person_counter;
12: call setName(p, sBear_Grylls);</pre></td></tr>
</table>

Рис. 2. Пример Java программы и соответствующей формальной модели

Описанный подход имеет недостатки – размер массива ограничен, при удалении объектов в массивах образуются «пустоты», при сравнении состояний модели играет роль не только содержимое объектов, но и порядок их создания, что влечет ненужный перебор. Эти проблемы будут решаться на уровне алгоритмов генерации путей поведения модели. Так, процедура сравнения двух состояний будет учитывать эту особенность, применяя методы симметрий для индексов, представляющих объекты.

Фильтрация кода. Необходимость фильтрации на этапе построения формальной модели продиктована спецификой промышленных проектов. Так, большие объемы и рост количества состояний могут быть уменьшены за счет кода, не влияющего на исследуемое поведение. К основным источникам такого кода можно отнести функциональность, не интересующую заказчика (например, устаревший пользовательский интерфейс), вычисление неактуальных данных, а также недостижимый код. Фильтрация выполняется аналогично построению слоев (slicing) – имея множество необходимых переменных, определяются статические связи с другими переменными, а оставшиеся удаляются из рассмотрения вместе с кодом, не содержащим необходимых и связанных с ними переменных.

Построим множество A необходимых переменных по следующему алгоритму:

1. Инициализируем A множеством переменных, находящихся в output-событиях (наблюдаемые переменные).

2. Для каждой переменной $v \in A$, найдем все присваивания (assignment-события), где v является левой частью: v = expr. Для каждого такого присваивания:

a. добавим в A переменные из правой части expr и индексы из левой;

b. если данное присваивание находится в теле цикла или ветвления по условию (событие loop или decision), то добавим в A переменные из всех соответствующих условий до верхнего уровня вложенности.

3. Повторяем (2), пока A не перестанет расширяться.

Можно показать, что приведенный алгоритм построит верхнюю аппроксимацию поведения модели по отношению к свойствам достижимости критериев покрытия.

Определение типов. После предыдущих трансформаций в выражениях остаются переменные и массивы примитивных Java типов (числовые и булевы), а также перечислимые и строки. Действительно, ссылки на объекты преобразованы в целочисленные индексы, коллекции – в массивы, а к прочим классам, которые остались неинтерпретированными, применена абстракция. Переменные будем преобразовать в атрибуты формальной модели с типами int, real, bool, перечислимыми и массивами согласно следующему алгоритму:

1. Для каждого атрибута соберем все предикаты и значения из правых частей присваиваний и предикатов равенства (Java операции "==", и "!=" и метод equals).

2. Анализируя каждый предикат, сгруппируем атрибуты по типам и объединим группы по транзитивности. Например, атрибуты из предикатов a > b [i] и b[j] := c будут сгруппированы так: {a, b, c}, {i}, {j}.

3. Для каждой группы атрибутов определим тип следующим образом:

   a. если среди всех предикатов нет арифметических операций над атрибутами этой группы и ни один атрибут не является индексом массива, то:

      i. из множества всех значений атрибутов группы сформируем новый перечислимый тип, применяя простые синтаксические преобразования для строк, символов и чисел ("Bear Grylls" –> sBear_Grylls, ' ' –> c32, 2016 –> i2016); если тип содержит всего два элемента, полученных из значений true/false или 1/0, то назначим группе тип bool, иначе добавим элемент, означающий отрицание всех значений, и назначим построенный перечислимый тип;

      ii. иначе: назначим группе тип int или real, в зависимости от значений.

Во время синтаксических преобразований имен и значений атрибутов отдельно сохраняются исходные данные, чтобы иметь возможность представлять сгенерированные тесты в терминах исходного кода.

Для уменьшения объемов формальных моделей производятся некоторые оптимизации. Например, вместо вызовов типичных для Java однострочных get- и set-методов, выполняется прямая подстановка соответствующих атрибутов.

События графа в формальной модели представлены переходами, которые в свою очередь имеют пред- и постусловия, выполняются атомарно за конечное время; их семантика аналогична охраняемым командам Дейкстры. Формальные определения модели приведены в [6, 12, 13].

Генератор тестовых сценариев осуществляет поиск поведения в соответствии с критерием покрытия (опционально statement, control-flow branch, def-use branch [5], а также покрытие сценариев [14, 15]). Так как перед нами не стоит задача верификации, мы можем отказаться от использования трудоемких солверов, заменив их примитивными операциями сравнения и присваивания для числовых типов и несколько более сложными для перечислимых. Атрибуты перечислимого типа будем кодировать двоичным вектором, при этом операция сравнения будет сводиться к нахождению пересечения двух векторов (такое упрощение в некоторых случаях может привести к ложным трассам, поэтому в конце работы мы будем проверять их традиционным символьным моделированием и уточнять модель в случае необходимости).

Редукция. Главной проблемой на этом этапе является эффект комбинаторного взрыва состояний модели. Наряду с существующими традиционными [10] методами редукции пространства поиска мы используем оригинальный метод динамической абстракции [12], и некоторые его оптимизации [13]. Основой этого метода служит алгоритм определения факта избыточности некоторых составляющих состояния модели по отношению к свойству достижимости, и построения соответствующего абстрактного состояния. Пусть задано конечное множество атрибутов $A = \{v_1, v_2, ..., v_n\}$ и пусть также для каждого атрибута $v_i \in A$ определена конечная область допустимых значений $D(v_i)$. Конкретное состояние модели включает в себя значения всех атрибутов: $q_c = \{\bigcup_{0 \le i \le |A|} (v_i = d_i) \,|\, v_i \in A \wedge d_i \in D(v_i)\}$, тогда как абстратное состояние включает в себя некоторое подмножество $A_{q_a} \subset A$ атрибутов, которым, свою очередь, сопоставляется множество конкретных значений: $q_a = \{\bigcup_i (v_i : u_i) \,|\, v_i \in A_{q_a} \wedge u_i \subseteq D(v_i)\}$. Такая абстракция определяет классы эквивалентности: два состояния s_1 и s_2 будут считаться одинаковыми, если их соответствующие абстрактные состояния совпадают, т.е. $Abs(s_1) = Abs(s_2)$. При этом множества трасс (а значит и достижимость критерия покрытия) абстрактной и конкретной модели совпадают [12]. В данной работе проверяемые свойства сводятся к проверке достижимости индивидуальных целей (определяется критерием покрытия, например, строка кода или def-use ветвь). Такая постановка задачи позволяет усовершенствовать построение абстракции, привязав функцию абстрагирования к выбранной цели покрытия. Текущее состояние s будем считать неперспективным, если из него не достижима ни одна из актуальных целей, т.е. $goal \in G \Rightarrow Abs^{goal}(s) = q_a \wedge q_a \in visited$. Здесь $visited$ – множество построенных ранее состояний, а G – множество актуальных (еще не покрытых) целей. Алгоритм поиска будет интерпретировать такие состояния как терминальные. Еще одно преимущество разбиения на индивидуальные цели – возможность формировать абстрактное состояние с учетом верхней аппроксимации – достаточно рассматривать только обратный слой

(backward slice) для выбранной точки. Описанная модификация носит ключевой характер для существенного сокращения пространства поиска. Подробное изложение заслуживает отдельного рассмотрения и выходит за рамки данной статьи.

Пролонгация. При нахождении трассы, покрывающей новый элемент, она по возможности пролонгируется до использования выполненных присваиваний, что обеспечит связность и осмысленность теста, а также позволит строить предположения (assertions) для дополнительных проверок.

При построении трассы использовалась техника так называемой «ленивой инициализации». Так, для каждого атрибута в модель добавлен признак инициализации (вспомогательный атрибут). При осуществлении переходов, значения атрибутов будут использоваться в соответствии с их инициализацией. Если признак установлен в истину, то это будет означать, что значение уже вычислено и его можно использовать. В противном случае, перед использованием значение атрибута необходимо вычислить и установить в истину признак инициализации. Процесс вычисления зависит от использования. Например, предикат $v = enum2$ установит вектор значений атрибута v перечислимого типа $T:\{enum1, enum2, enum3, enum4\}$ в (0,1,0,0), а $\neg(v = enum2)$ соответственно в (1,0,1,1). Для вычисления конкретных значений, удовлетворяющих условие теста (и, аналогично, значений параметров сигналов), будем двигаться по трассе в обратном направлении – от финального состояния к начальному. Для каждого атрибута зафиксируем произвольное (из множества доступных) значение и будем продвигать его в соответствии с переприсваиваниями и ограничениями предусловий. Так как между присваиваниями диапазон значений атрибута не расширяется, в итоге будет вычислено искомое значение. Достаточное условие теста составляют те атрибуты, значение которых использовалось без соответствующего присваивания. Пример программы, трассы и соответствующего условия теста приведен на рис.3.

<table>
<tr>
<td>public void fun(T v){
    T p;
    if(v != T.enum2){
       if(v != T.enum3)
         sendA(v);
    }
    p = v;
    if(p == T.enum4)
       sendB(p);
}</td>
<td>if(v != T.enum2)
if(v != T.enum3)
sendA(v)
p = v
if(p == T.enum4)
sendB(p)</td>
<td>v = T.enum4;
sendA(T.enum4)
sendB(T.enum4)</td>
</tr>
</table>

Рис. 3. Пример Java программы (а), трассы(б),
условия теста и значений параметров(в)

Минимизация тестового набора. По завершении генерации созданный набор минимизируется путем удаления трасс, избыточных по отношению к критерию покрытия. Так, построенные трассы перепроверяются в обратном (по отношению к генерации) порядке на предмет обнаружения дублирующих покрытий. На практике такой метод сокращает исходный набор тестов на 10–20 %.

Выводы

Предложен новый метод порождения тестовых наборов по исходному коду для языка Java. Среди отличительных особенностей выделяются метод построения и сокращения формальной модели, работа с неинтерпретированными функциями, а также эффективные методы редукции пространства поиска. Несмотря на неполную автоматизацию (необходимо написание mock-функций для неинтерпретированных

языковых конструкций), применение метода на практике позволило получить перспективные результаты: на примере промышленного кода, размер которого превышает 105 строк, построена модель, состоящая из порядка 3 тыс. атрибутов и 12 тыс. переходов, при этом удалось сгенерировать ~1200 тестов, обеспечивающих 94% покрытие, за время менее 5 минут. Несмотря на довольно большую степень абстракции некоторых сложных арифметических выражений, деталей SQL запросов и т.п., полученные тесты оказались представительными для проверки бизнес-логики. Более того, использование методов [6–8] обеспечило проверку правильности тестов и позволило выразить условие для теста в виде формулы, определяющей диапазон всех возможных конкретных значений для дополнительного анализа кода. Также стоит отметить, что метод построения формальной модели дал возможность обработать неполный промышленный код, исполнение которого не представлялось возможным.

Описанный метод усовершенствует работы [9, 12–15]: так, на несколько порядков увеличена эффективность процесса подстановки конкретных значений; во многих случаях экспоненциально снижено количество состояний (в частности, для примера на рис.1 показана линейная зависимость), добавление критерия покрытия def-use ветвей позволило существенно повысить качество тестов.

Литература

1. Mahadik P., Thakore D. Survey on Automatic Test Data Generation Tools and Techniques for Object Oriented Code // Int. J. of Research in Computer and Communication Engineering. –2016. –Vol 4. –P. 357–364.
2. Cseppento L., Micskei Z. Evaluating Symbolic Execution-based Test Tools // In Proc. of IEEE Int. Conf. on Software Testing, Verification and Validation (ICST). –2015. –P. 1–10.
3. Pasareanu C., Visser W., and oth. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis // Automated Software Engineering. –2013. –Vol. 20. –N3. –P.391–425.
4. Fraser G., Arcuri A. EvoSuite at the SBST 2016 tool competition // Proc. of the 9th Int. Workshop on Search-Based Software Testing. –2016. –P. 33–36.
5. Frankl P., Weyuker E. An applicable family of data flow testing criteria // IEEE Transactions on Software Engineering. –1988. –Vol 4. –P. 1483–1498.
6. Летичевский А.А., Годлевский А.Б. и др. Свойства предикатного трансформера системы VRS // Кибернетика и системный анализ. –2010. –№4. –С. 3–16.
7. Потиенко С.В. Методы прямого и обратного символьного моделирования систем, заданных базовыми протоколами // Проблемы программирования. – 2008. – № 4. – С. 39–45.
8. Годлевский А.Б., Потиенко С.В. Обратная трансформация формул в символьном моделировании: от результата к исходной формуле // Проблемы программирования. – 2010. – № 2–3. – С. 363–368.
9. Kolchin A., Letichevsky A., Peschanenko V., Drobintsev P., Kotlyarov V. An approach to creating concretized test scenarios within test automation technology for industrial software projects // Automatic Control and Computer Sciences. –2013. –Vol. 47(7). –P. 433–442.
10. Jhala R., Majumdar R. Software model checking // ACM Comput. Surv. – Vol.41(4). – 2009. – 54 P.
11. [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/javase/8/docs
12. Колчин А.В. Автоматический метод динамического построения абстракций состояний формальной модели // Кибернетика и системный анализ. – 2010. – № 4. – С. 70–90.
13. Колчин А.В. Метод редукции анализируемого пространства поведения при верификации формальных моделей распределенных программных систем // Искусственный интеллект. –2013. – №4. –С. 113–126.
14. Летичевский А.А., Колчин А.В. Генерация тестовых сценариев на основе формальной модели // Проблемы программирования. – 2010. – № 2–3. – С. 209–215.
15. Колчин А.В., Дробинцев П.Д., Котляров В.П. Метод генерации тестовых сценариев в среде инсерционного моделирования // Управляющие системы и машины. –2012. – № 6. – С. 43–48,63.

Literatura

1. Mahadik P., Thakore D. Survey on Automatic Test Data Generation Tools and Techniques for Object Oriented Code // Int. J. of Research in Computer and Communication Engineering. –2016. –Vol 4. –P. 357–364.
2. Cseppento L., Micskei Z. Evaluating Symbolic Execution-based Test Tools // In Proc. of IEEE Int. Conf. on Software Testing, Verification and Validation (ICST). –2015. –P. 1–10.

3. Pasareanu C., Visser W., and oth. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis // Automated Software Engineering. –2013. –Vol. 20. –N3. –P.391–425.
4. Fraser G., Arcuri A. Whole test suite generation // IEEE Transactions on Software Engineering. – 2013. – Vol. 39. –N2. –P. 276–291.
5. Frankl P., Weyuker E. An applicable family of data flow testing criteria // IEEE Transactions on Software Engineering. –1988. –Vol 4. –P. 1483–1498.
6. Letichevskiy A.A., Godlevskiy A.B. i dr. Svoystva predikatnogo transformera sistemyi VRS // Kibernetika i sistemnyiy analiz. –2010. –#4. –S. 3–16.
7. Potienko S.V. Metodyi pryamogo i obratnogo simvolnogo modelirovaniya sistem, zadannyih bazovyimi protokolami // Problemyi programmirovaniya. – 2008. – # 4. – S. 39–45.
8. Godlevskiy A.B., Potienko S.V. Obratnaya transformatsiya formul v simvolnom modelirovanii: ot rezultata k ishodnoy formule // Problemyi programmirovaniya. – 2010. – # 2–3. – S. 363–368.
9. Kolchin A., Letichevsky A., Peschanenko V., Drobintsev P., Kotlyarov V. An approach to creating concretized test scenarios within test automation technology for industrial software projects // Automatic Control and Computer Sciences. –2013. –Vol. 47(7). –P. 433–442.
10. Jhala R., Majumdar R. Software model checking // ACM Comput. Surv. – Vol.41(4). – 2009. – 54 P.
11. [Elektronnyiy resurs]. – Rezhim dostupa: https://docs.oracle.com/javase/8/docs
12. Kolchin A.V. Avtomaticheskiy metod dinamicheskogo postroeniya abstraktsiy sostoyaniy formalnoy modeli // Kibernetika i sistemnyiy analiz. – 2010. – # 4. – S. 70–90.
13. Kolchin A.V. Metod reduktsii analiziruemogo prostranstva povedeniya pri verifikatsii formalnyih modeley raspredelennyih programmnyih sistem // Iskusstvennyiy intellekt. –2013. – #4. –S. 113–126.
14. Letichevskiy A.A., Kolchin A.V. Generatsiya testovyih stsenariev na osnove formalnoy modeli // Problemyi programmirovaniya. – 2010. – # 2–3. – S. 209–215.
15. Kolchin A.V., Drobintsev P.D., Kotlyarov V.P. Metod generatsii testovyih stsenariev v srede insertsionnogo modelirovaniya // Upravlyayuschie sistemyi i mashinyi. –2012. – # 6. – S. 43–48,63.

RESUME

A. Kolchin, S. Potiyenko

A method of test data generation from source code of Java programs

A new method for test data generation for Java programs is proposed. The method is based on the original effective algorithms for formal model construction, search space reduction, approximation and abstract interpretation. While the test generation process is not fully automated (mock-functions needed for some non-interpreted Java constructions), applying of the method in practice gives perspective results: from 105 lines of a real industrial code we have constructed a formal model which contains ~3 thousands of attributes and ~12 thousands of guarded commands; our tool generates ~1200 tests, which makes 94% coverage in 5 minutes. In spite of high level of abstraction from complex math. functions and detailed SQL queries, the tests obtained was representative enough to cover the business logic. Moreover, by using of backward predicate transformer, our tool produces formula of test condition, which includes range of all admissible concrete values for each generated test. The method of constructing a formal model made it possible to generate tests for incomplete code, which execution was not possible. In order to generate tests for inter-procedural interaction, we use the following scheme: the whole model is considered as a big endless loop, which consists of alternative calls of all public methods. Then we apply def-use branches coverage criterion. Such approach has significantly increased the semantic quality of generated tests: traces become more intelligent due to meaningful relationship between events. In many cases the described state-space reduction method allowed to avoid exponential explosion, which is a crucial problem on the way of integration of formal methods into software industry.

Надійшла до редакції 02.09.2016